Preliminary Interface Control Document for the Conflict Detection And Resolution Function of the Airborne Operational Planner System

November 2000

Prepared for:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA

Prepared by:

CSSI Inc.
600 Maryland Ave., SW
Suite 890
Washington, DC 20024

Titan Corp.
Burlington, MA

Author Contact Information:

Stephane Mondoloni, Ph.D.   (202) 863-2175   smondoloni@cssiinc.com

Table of Contents

List of Figures

1 Scope

1.1 Identification

This Interface Control Document (ICD) describes the interfaces to the Conflict Detection and Resolution (CD&R) component of the Airborne Operational Planner (AOP). The following interfaces are described for the CD&R component.

1. The interface between the Conflict Detection and Resolution Function (CD&R) and the Airborne Operational Planner (AOP).
2. The interface between the CD&R and an external Flight Rules (FR) function. The flight rules function accepts information from CD&R and responds with conflict constraints based upon specified rules. These constraints may specify whether the aircraft should move and what some constraints on maneuvers should be.
3. The interface between the CD&R and the FMS pre-processor.
4. The interface between the CD&R and the Constraint Manager (CM).
5. The interface between CD&R and User Input Function.
6. The interface between CD&R and the Long Term Optimization Function.
7. The interface between CD&R and the Crew Notification Function.

1.2 System Overview

The Conflict Detection and Resolution function is a software component of the Airborne Operational Planner responsible for the detection and resolution of conflicts due to hazards such as traffic hazards, Special Use Airspace (SUA), inclement weather and terrain. The CD&R function defined in this document describes an "intent-based" CD&R function based upon the intended flight trajectory of the own-ship and neighboring aircraft. The overall CD&R requirements are defined in the document Software Specification for the Conflict Detection and Resolution Function of the Airborne Operational Planner.

In order to provide the specified functionality, the CD&R function must interface with a variety of external functions and databases. Each of the interfaces serves a particular role as defined below.

a) AOP to CD&R – This interface is the primary interface between the AOP and the CD&R function. The AOP submits own-ship flight plan information, own-ship state information, area hazard descriptions, and intruder flight trajectory information. Flight plan information is allowed to contain waypoint constraints. If required, the AOP may also submit boundary constraints such as required times of arrival at a boundary. Upon completion of conflict resolution, the CD&R function responds to the AOP with an indication of success or failure.

b) CD&R to flight rules – The CD&R function interfaces with a flight rules (FR) function for the purpose of determining which aircraft in a conflict is required to execute an avoidance maneuver. The rules can optionally be used to obtain constraints on maneuvers. These constraints can be used to provide heuristic conflict resolution maneuvers, or to provide for standardized cooperative maneuvers.

c) CD&R and FMS Preprocessor – The CD&R function interfaces with an FMS preprocessor for the purpose of obtaining a flight trajectory from a flight plan. The flight plan may contain a series of constraints. The FMS is expected to respond to the submission of a flight plan with a flight trajectory.

d) CD&R and Constraint Manager – The CD&R function interfaces with a Constraint Manager function. The CD&R function calls the Constraint Manager function in the event that CD&R is unable to obtain a solution given all of the specified constraints and hazards. The Constraint Manager (CM) function obtains a flight plan, trajectory, hazard and conflict information from the CD&R function. The Constraint Manager provides the CD&R function with a combination of flight plan constraint relaxation, hazard prioritization and flight rules modification in order to allow the CD&R to obtain a resolution maneuver.

e) CD&R and User Input Function – The CD&R function interfaces with a User Input function. The purpose of this interface is to allow the crew to input user-preferences (e.g. minimum time resolution), maneuver constraints (e.g. lateral maneuvers should be obtained), specific flight plans during manual resolution mode, and specification of the resolution mode desired. The interface with a User Input function is to be indirect through the AOP calling function.

f) CD&R and Long Term Optimization Function – The CD&R function interfaces with the Long Term Optimization Function in order to obtain an optimized flight plan and trajectory for conflict detection. The Optimization function should also be capable of submitting to CD&R an indicator that no optimized flight plan is available.

g) CD&R and Crew Notification Function – The CD&R function interfaces with a Crew Notification Function in order to provide feedback to the crew. Conflict information, area hazards, and status information (such as right-of-way) are all submitted to the Crew Notification Function. The interface with the Crew Notification Function will be indirect through the AOP calling function.

1.3 Document Overview

The purpose of this document is to describe the interfaces for the Conflict Detection and Resolution function. Each interface is described in a separate subsection beginning at Section 3.2.

2 Applicable Documents

2.1 Government Documents

The following documents of the exact issue shown form a part of this document to the extent specified herein.

Autonomous Operations Planner – System Requirements and High Level Design-January, 2000.

2.2 Non-Government Documents

The following documents of the exact issue shown form a part of this document to the extent specified herein.

Preliminary Software Specification for the Conflict Detection and Resolution Function of the Airborne Operational Planner – February, 2000.

Airborne Operational Planner (AOP) Conflict Resolution Algorithm Description, April, 2000.

FMS Prediction, Draft v.8, Sam Liden, July, 2000.

ACE online manual at:

http://www.cs.wustl.edu/~schmidt/ACE_wrappers/man/acewindex.html

3 Interface Description

3.1 Interface Diagrams

Figure 3.1-1 illustrates the connectivity of the CD&R function to external functions. Functions that are expected to operate concurrently are indicated with shaded areas. Thus, the User Input Function is anticipated to be operating concurrently with the CD&R function and data is only expected to flow from the User Input Function to the CD&R function. Heavy arrows indicate the direction of the initial call and light arrows indicate the direction of a response.

The CD&R function is decomposed into two distinct components, one that is expected to operate concurrently with AOP, CNF and the User Input Function, and one that is not expected to be concurrent.

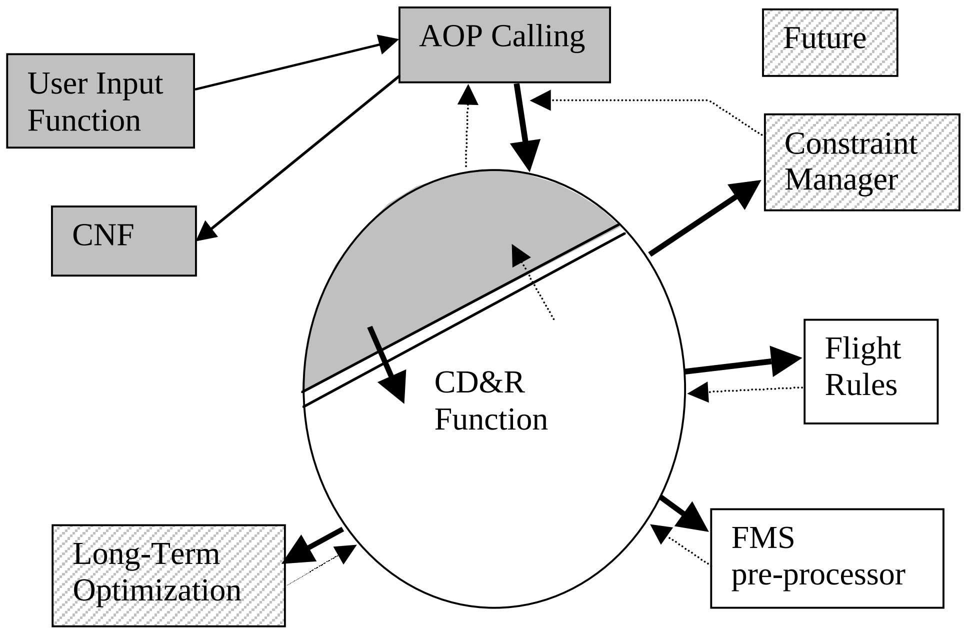

Figure 3.1-1 CD&R Function Interfaces

3.2 CD&R and AOP calling function

The CD&R function and the AOP calling function will exchange data as specified in this section.

3.2.1 Protocol

The CD&R and the AOP calling function operate concurrently. Data can be received asynchronously by the CD&R function from the AOP calling function. Data can be sent asynchronously from the CD&R function to the AOP calling function.

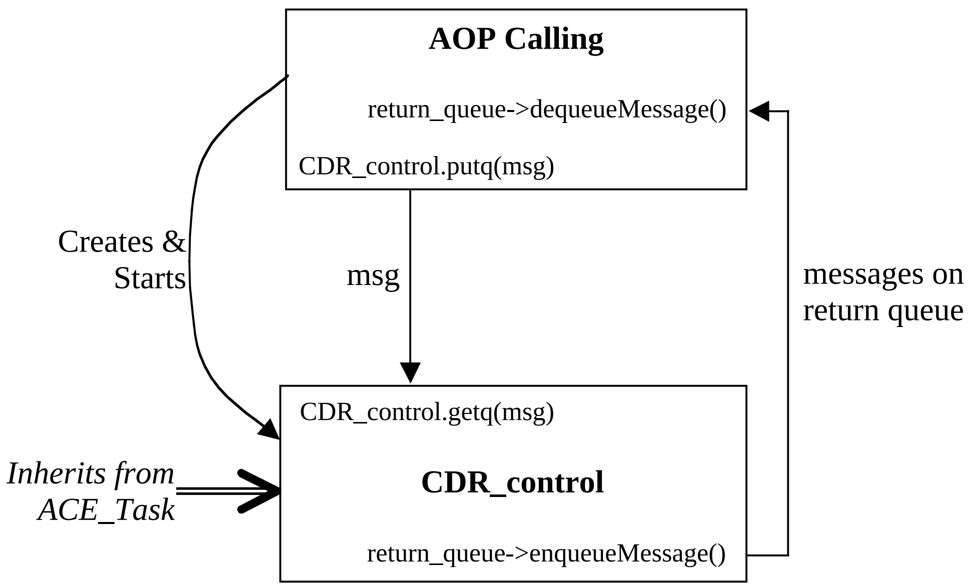

Figure 3.2.1-1 Schematic of message passing between AOP Calling functions and CD&R Control function.

Classes defined by the Adaptive Communications Environment (ACE) are used in defining the protocol between the AOP calling function and the CD&R function. Figure 3.2.1-1 describes at a high level the interaction between the AOP Calling function and the CD&R control function. This interaction can be broken into three separate types.

- The AOP Calling function creates and starts the CD&R control function. The CD&R control function will subsequently operate in a separate thread from the AOP Calling function. Upon starting the CD&R control function, the AOP calling function supplies the CD&R function with a pointer to a return queue to be used for message passing.
- The AOP Calling function will send messages to the CD&R control function through the CD&R control queue.
- The CD&R control function will return messages to the AOP Calling function via the return queue supplied.

3.2.1.1 Creating and Starting the CD&R Control Task

It is assumed that a TaskCDRControl class is instantiated and started by AOP. This class inherits from the ACE_Task and (has a svc method which) operates as an independent thread from the AOP calling function. AOP can instantiate a TaskCDRControl class as follows:

```
TaskCDRControl CDR_control;
```

The above task is started by invoking the start method, which starts a separate thread for the CDRControl task.

```
int TaskCDRControl::start (ReturnQueue *)
```

Note that the above will return a value of –1 if the method cannot be started. A pointer to a ReturnQueue is provided in order for messages to flow back to the AOP calling function. In order to receive messages from the CD&R function, the AOP needs to instantiate a ReturnQueue class and provide the address to the start method of the TaskCDRControl class.

3.2.1.2 Messages from AOP to CD&R

Messages are passed from the AOP calling function to the CD&R function by invoking the putq method of the TaskCDRControl class as shown below.

```
int TaskCDRControl::putq(Message_Block *);
```

Messages from the AOP calling function to the CD&R function are encoded into a Message_Block class inheriting from the ACE_Message_Block class.

The AOP calling function is expected to provide the CD&R function with messages described below. The message names containing those data elements are shown in parentheses).

- Own-ship flight plan with constraints (FLIGHT_PLAN)
- Own-ship state information (STATE)
- Area hazard description (AREA_HAZARD)
- Area hazard to be removed (REMOVE_AREA_HAZARD)
- Intruder aircraft flight trajectories (TRAFFIC)
- Intruder flight trajectories to be removed (REMOVE_TRAFFIC)
- Boundary Constraints (BOUNDARY_CONSTRAINT)

The AOP calling function will additionally supply the following information assumed to originate from a User Input function. The message names are shown in parentheses.

- Flight plans for manual resolution or trial planning (MANUAL_FP_RES)
- Selection of Automatic resolution mode (FORCE_AUTOMATIC)
- Request of next resolved flight plan (REQUEST_RESOLVED)
- Acceptance of resolution (ACCEPT_RESOL)
- User-supplied maneuver constraints (MANEUVER_CONSTRAINT)
- User-supplied maneuver preferences (MANEUVER_PREFERENCE)

The above messages are encoded into the Message_Block class defined below.

```
class Message_Block : public ACE_Message_Block

{
public:
   typedef ACE_Message_Block inherited;

  // Constructors
  Message_Block(size_t );
  Message_Block(size_t , StateCR*);
  Message_Block(size_t , FlightPlan*);
  Message_Block(size_t , struct Ownship*);
  Message_Block(size_t , struct Intruder *);
  Message_Block(size_t , struct AreaHazard*);
  Message_Block(size_t , int, BoundaryConstraint*);
  Message_Block(size_t , ManeuverList *);
  Message_Block(size_t , ManeuverPreference*);
  Message_Block(size_t , struct Conflict*, struct AreaConflict*);

  // Destructor
  ~Message_Block();

  // Accessor Methods
  StateCR *state(void);
  FlightPlan* flightPlan(void);
  struct Ownship* getTrajectory(void);
  struct Intruder* getIntruder(void);
  struct AreaHazard* getAreaHazard(void);
  BoundaryConstraint* getBoundaryConstraint(void);
  int getNumberOfBoundaries(void);
  ManeuverList* getManeuverList(void);
  ManeuverPreference getManeuverPreference(void);
  struct Conflict* getConflictData(void);
  struct AreaConflict* getAreaConflictData(void);
protected:
 (… protected data in here, not relevant to ICD)
};
```

Messages are encoded into the message block by invoking the appropriate constructor given the data being passed. The appropriate constructor for each message type is defined in the table below.

Constructor	Message Type
Message_Block(size_t);	FORCE_AUTOMATIC REQUEST_RESOLVED ACCEPT_RESOL
Message_Block(size_t , StateCR*);	STATE
Message_Block(size_t , FlightPlan*);	FLIGHT_PLAN MANUAL_FP_RES
Message_Block(size_t , struct Ownship*);	OWN_TRAJECTORY
Message_Block(size_t , struct Intruder *);	TRAFFIC REMOVE_TRAFFIC
Message_Block(size_t , struct AreaHazard*);	AREA_HAZARD REMOVE_AREA_HAZARD
Message_Block(size_t , int, BoundaryConstraint*);	BOUNDARY_CONSTRAINT
Message_Block(size_t , ManeuverList *);	MANEUVER_CONSTRAINT
Message_Block(size_t , ManeuverPreference*);	MANEUVER_PREFERENCE

An example is provided below to describe how a “TRAFFIC” message would be placed onto the CDR_Control queue.

```
// A current intruder is instantiated
struct Intruder* cur_intruder = new struct Intruder();

// Code in here will assign data to the intruder

// The intruder is placed onto a message block with enough space // for text (128)
Message_Block *message = new Message_Block(128, cur_intruder);

//Put text message into block and move to end of message
ACE_OS::sprintf(message->wr_ptr (), "TRAFFIC");
message->wr_ptr(strlen(message->rd_ptr ())+1);

//Add the message to CDR control queue and error check
if (CDR_control.putq(message) == -1)
{
      // Error message in here
}
```

The CD&R function will retrieve the above messages through the getq method of the ACE_Task class. Data within the message is retrieved using the rd_ptr method of the

ACE_Message_Block class, and using the accessor methods of the Message_Block class defined above.

3.2.1.3 Messages from CD&R to AOP

Messages are passed from the CD&R control class to the AOP calling function by placing them onto the return queue. This is achieved by invoking one of the following methods.

```
int ReturnQueue::enqueueMessage(const char*)
int ReturnQueue::enqueueMessage(const char*, FlightPlan*)
int ReturnQueue::enqueueMessage(const char*, struct Conflict* , struct AreaConflict*)
```

A simple text message uses the first method, a message including a flight plan uses the second method and a message supplying conflict information will use the third method.

The AOP calling function may retrieve the above messages by invoking the following method. Note that the message text is placed onto a character string and all other data is obtained by passing pointer addresses.

```
int ReturnQueue::dequeueMessage(char [50], FlightPlan** , struct Conflict**, struct
AreaConflict**)
```

In response to many of the messages from the AOP to the CD&R function, the CD&R function may send messages back to the AOP calling function. Many of these messages are eventually destined for display to the flight crew. The following table describes the messages that are appropriate responses to original messages.

Outgoing Message (AOP to CD&R)	**Response Message (CD&R to AOP)**	**Significance**
ACCEPT_RESOL	ERROR_NO_RES INITIALIZING:_NO_RES ERROR_CANNOT_ACCEPT_RES	No active resolution task In initialization mode Unable to stop resolution
Any message initiating CD&R	NO_CONFLICT CONFLICT THEY_MOVE WE_MOVE	No conflict found Conflicted flight Flight need not move Flight required to move
REMOVE_AREA	ERROR_CANNOT_STOP_RES ERROR:CANNOT_REMOVE_AREA ERROR:AREA_NOT_PRESENT_TO_REMOVE	No active resolution task No hazards to remove Hazard not found
REMOVE_TRAFFIC	ERROR_CANNOT_STOP_RES ERROR:CANNOT_REMOVE_TRAFFIC ERROR:TRAFFIC_NOT_PRESENT_TO_REMOVE	No active resolution task No hazards to remove Hazard not found
REQUEST_RESOLVED	ERROR_NO_RES ERROR_CANNOT_REQUEST_RES INITIALIZING:_NO_RES NOT_RESOLVED_YET RESOLVED_FP LAST_RESOLVED_FP	No active resolution task Unable to enqueue msg In initialization mode No resolution found yet Resolution FP provided Last resolved FP in list

AREA_HAZARD BOUNDARY_CONSTRAINT FLIGHT_PLAN TRAFFIC STATE	ERROR_CANNOT_REQUEST_RES	Unable to enqueue msg
FORCE_AUTO	INITIALIZING_CANNOT_FORCE_AUTO ERROR_CANNOT_REQUEST_RES	In initialization mode Unable to enqueue msg
MANEUVER_CONSTRAINT	INITIALIZING:MANEUVER_STORED NO_CONFLICT_FOR_MANEUVER	Store maneuver when in initialization mode No conflict exists for this maneuver constraint
MANUAL_FP_RES	INITIALIZING: NO MANUAL RES CONFLICT_FREE ***FLID*** CONFLICTED ***FLID***	In initialization mode ***FLID*** is conflict-free ***FLID*** is conflicted
Completed resolution	RESOLVED_FP RESOLUTION_NOT_FOUND	Resolved FP is provided No resolution was found

The above messages are all placed in the ReturnQueue class along with appropriate data. The ReturnQueue class is described below.

```
class ReturnQueue
{
public:
  // Constructor creates a queue
  ReturnQueue();

  // Enqueuing methods
  int enqueueMessage(const char *);
  int enqueueMessage(const char *, FlightPlan *);
  int enqueueMessage(const char *, struct Conflict*, struct AreaConflict*);

  // Dequeues messages one at a time and returns pointers (if applicable)
  int dequeueMessage(char [], FlightPlan **, struct Conflict **, struct AreaConflict**);

private:
  ACE_Message_Queue<ACE_MT_SYNCH> *mq_;     // Underlying queue
};
```

Most message types will enqueue just a text message using the first constructor. In this case, the dequeueMessage method returns pointers to NULL for flight plan and conflict data. The enqueueMessage constructor with the flight plan is only called for the following messages: RESOLVED_FP and LAST_RESOLVED_FP. The constructor with conflict data is only called for the CONFLICT and CONFLICTED messages.

3.2.2 Priority

Data transferred from the AOP calling function to the CD&R function consists of data such as hazard descriptions (area and intruder), flight plans, flight trajectories, and aircraft-state information. It is anticipated that certain data elements will have higher priority than others (closer hazards, active own-ship trajectories versus provisional trajectories). Data transferred between the AOP calling function and the CD&R function may contain priority information for use by the end application. ***Requirements for this priority information have not yet been defined. The current CD&R build does not incorporate priority information.***

3.2.3 Data Elements

All of the messages described above contain data types defined in this and subsequent sections.

3.2.3.1 Own-ship Flight Plan Data Description

A flight plan is defined through the following class definition.

```
class FlightPlan
{
public:

        // Constructor
        FlightPlan();

        // Destructor
        ~FlightPlan();

  double time_stamp;
  char *flight_number;
  struct AirportType *origin;
  struct AirportType *destination;
  struct RunwayType *departure_runway;
  struct RunwayType *arrival_runway;
  double cost_index;
  struct AltitudeList* cruise_altitude;        // Implemented as a double for initial CD&R
  double required_climb_cas;
  double required_climb_mach;
  double required_cruise_speed;
  double required_descent_mach;
  double required_descent_cas;
  int number_of_waypoints;
  Waypoint *waypoint;
};
```

Note that the cruise altitude is implemented as a double for the initial CD&R capability since the trajectory generation function currently supports only a single altitude. The elements of the class are defined in the following table.

Data Element	Description	Type / Units	Range
time_stamp	A time stamp indicating the time at which this flight plan was created. Note that changes in PPOS as the first point in the flight plan result in an update to the timestamp.	double / minutes	based on AOP system clock range
flight_number	A string tracking the flight number (e.g. AAL001).	char *	ASCII string
origin	A pointer to origin airport data.	struct AirportType *	Null or pointer to valid airport data
destination	A pointer to destination airport data.	struct AirportType *	Null or pointer to valid airport data
departure_runway	A pointer to departure runway data	struct RunwayType *	Null or pointer to valid runway data
arrival_runway	A pointer to arrival runway data	struct RunwayType *	Null or pointer to valid runway data
cost_index	Cost Index for flight	double	0 - 999
cruise_altitude	A sequential list of cruise altitudes for the flight. The first element in the list is the initial cruise altitude and other elements represent altitudes for step climbs and descents	struct AltitudeList *	Pointer to valid list. (never NULL)
required_climb_cas	Desired CAS for the constant CAS portion of climb	double / knots	>0
required_climb_mach	Desired Mach number for the constant Mach portion of climb	double	>0
required_cruise_speed	Desired cruise speed. A mach number is assumed for speeds less than 50. CAS is assumed for speeds greater than 50.	double / knots or Mach	>0
required_descent_mach	Desired Mach number for a constant Mach portion of descent	double	>0
required_descent_cas	Desired CAS for a constant	double / knots	>0

	CAS portion of descent		
number_of_waypoints	Number of waypoints to follow in the list of waypoints.	int	>0
waypoint	Pointer into a list of valid waypoints.	Waypoint*	Pointer to valid list or NULL

The above data description requires that several data types be defined, these are defined below.

3.2.3.1.1 Airport Type

The airport type is defined through the following data structure. This data structure is currently a simple placeholder and is expected to be modified.

```
typedef struct AirportType {
  char *name;
  double latitude;
  double longitude;
} AirportType;
```

Data Element	**Description**	**Type / Units**	**Range**
name	Name of the airport (ICAO)	char *	ASCII string
latitude	Airport reference latitude (North is positive)	double / radians	$(-\pi, \pi)$
longitude	Airport reference longitude (East is positive)	double / radians	$(-\pi, \pi)$

3.2.3.1.2 Runway Type

The runway type is defined through the following data structure. The data structure is simply a placeholder for future runway information. Future runway information may include runway length, direction, airport reference, altitude, threshold location, slope, etc.

```
typedef struct RunwayType {
  char *name;
} RunwayType;
```

Data Element	**Description**	**Type / Units**	**Range**
name	Runway name (e.g. 22L)	char *	ASCII string

3.2.3.1.3 Altitude List Type

A list of altitudes represents a list of valid cruise altitudes for the flight and subsequent step climbs and descents. The altitude list is defined through the following doubly linked list data structure. The first element in the list represents the first cruise altitude.

```
typedef  struct AltitudeList
{
  double altitude;
  struct AltitudeList *next;
  struct AltitudeList *prev;
};
```

Data Element	Description	Type / Units	Range
altitude	Desired altitude	double / feet	0 – 99,999
next	Pointer to subsequent altitude step. Points to NULL if last in list	struct AltitudeList *	valid pointer to next list element, or NULL
prev	Pointer to previous altitude step. Points to NULL if first element	struct AltitudeList *	valid pointer to prior list element, or NULL

3.2.3.1.4 Waypoint Type

The waypoint type describes the waypoint data in some detail. The data structure defined below contains references to data types that will subsequently be defined.

```
class Waypoint
{
public:
        Waypoint();          // Constructor
        ~Waypoint();         // Destructor;

        char *identifier
        char *waypoint_name;
        double latitude;
        double longitude;
        Xyz xyz_location;
        Restriction time_restriction;
        Restriction speed_restriction;
        Restriction mach_restriction;
        Restriction altitude_restriction;
        struct AtmosphericType atmospheric;
        struct mcpState *requested_mcp;
        StateCR* forecast_state;
        double delta;
        int n_add;
        int original;
        int required;
        Waypoint *next;
        Waypoint *prev;
};
```

Data Element	Description	Type / Units	Range
identifier	An identifier for the waypoint	char *	ASCII string
waypoint_name	A name for the waypoint	char *	ASCII string
latitude	Latitude of the waypoint (N is positive)	double / radians	$(-\pi,\pi)$
longitude	Longitude of the waypoint (E is positive)	double / radians	$(-\pi,\pi)$
xyz	Unit vector corresponding to waypoint location	Xyz	valid XYZ structure
time_restriction	Structure representing RTA constraints at this waypoint.	Restriction/ bounds in minutes	
speed_restriction	Structure representing CAS constraints at this waypoint	Restriction/ bounds in knots	
mach_restriction	Structure representing MACH restrictions at this waypoint	Restriction	
altitude_restriction	Structure representing altitude constraints at this waypoint	Restriction/ bounds in feet	
atmospheric	Structure representing atmospheric data at this waypoint	struct atmosphericType	
requested_mcp	Pointer to a structure representing anticipated mode control panel (MCP) data at this waypoint	struct mcpState*	
forecast_state	Pointer to the nearest state data in the corresponding trajectory. Prior to trajectory generation this is set to NULL.	StateCR *	NULL or pointer to valid data
delta	Cross-track deviation from the original flight plan. (Used internally in CD&R, expect this to be set to 0 upon receipt of flight plan by CD&R.)	double / nmi	valid double
n_add	Number of off-track waypoints permitted after this one. (Used internally in CD&R, expect this to be set to 0 upon receipt of flight plan by CD&R.)	int	≥ 0

original	Flag indicating whether this point is part of the original flight plan (0 = no)	int	valid int
required	Flag indicating whether this waypoint is required as part of the flight plan (no = 0)	int	valid int
next	Pointer to the next waypoint in the flight plan. Set to NULL for last waypoint in flight plan.	Waypoint*	NULL or pointer to valid data
prev	Pointer to the previous waypoint in the flight plan. Set to NULL for first waypoint in flight plan.	Waypoint*	NULL or pointer to valid data

3.2.3.1.4.1XYZ Data Type

The Xyz data structure is used to represent the unit vector from the center of the earth to a specific location. The algorithm for conversion to this data structure is described in Section 4.2.2.1 of "Airborne Operational Planner (AOP) Conflict Resolution Algorithm Description", April, 2000. The data structure is defined below.

```
typedef struct {
  double x;
  double y;
  double z;
} Xyz;
```

Data Element	Description	Type / Units	Range
x	x – coordinate of unit vector	double	(-1, 1)
y	y – coordinate of unit vector	double	(-1, 1)
z	z – coordinate of unit vector	double	(-1,1)

3.2.3.1.4.2Restriction Data Type

Each flight plan restriction is described following the same data structure as defined below. Note that the units are based on the use of the restriction data type. For instance, a speed restriction will have units of knots for the upper and lower bounds.

```
enum Code {INACTIVE, AT, AT_OR_ABOVE, AT_OR_BELOW, BETWEEN};

typedef struct Restriction
{
        Code type;
        double lower;
        double upper;
        int required;
} Restriction;
```

Data Element	Description	Type / Units	Range
type	Definition of the type of constraint at this point. This data needs to be consistent with the lower and upper bounds defined below.	enum	INACTIVE AT AT_OR_ABOVE AT_OR_BELOW BETWEEN
lower	Lower bound of constraint	double	valid double
upper	Upper bound of constraint	double	valid double
required	Flag indicating whether this constraint is required to remain unchanged through the resolution process. (false = 0)	int	valid int

3.2.3.1.4.3Atmospheric Data Type

This data type is not expected to remain as it is currently defined. The current definition stems from the need to interface with Fastwin for testing purposes. CD&R does not require atmospheric data and is merely passing this data through. However, deep copies of waypoint data, and addition of waypoints within CD&R require some internal CD&R knowledge of this data type. The current atmospheric data element is defined below and is associated with an individual waypoint location.

```
typedef struct AtmosphericType {
  double altitude[5];
  double temp[5];
  double wind_speed[5];
  double wind_direction[5];
  double temp_dev[5];
} AtmosphericType;
```

Data Element	Description	Type / Units	Range
altitude[]	Array of altitudes at which the atmospheric data is defined .	double / feet	0 – 99,999
temp[]	Array of temperatures at the corresponding altitude above.	double / degrees C	>-273.15
wind_speed[]	Array of wind speeds at the corresponding altitudes above	double / knots	≥0
wind_direction[]	Array of wind direction from true north at the corresponding altitudes above	double / radians	(0,2π)
temp_dev[]	Array of temperature	double /	valid double

	deviations from ISA at the corresponding altitudes above	degrees C	

3.2.3.1.4.4 MCP State Data Type

The MCP State data is currently included as a placeholder in the event that future AOP functionality will desire to use the current MCP settings (and possibly forecast MCP settings) to determine alternate trajectories. Tracking this data through the CD&R function allows resolution to be performed on these types of trajectories in the future. A placeholder MCP data type is defined with no particular significance to the data.

```
typedef struct mcpState {
  double time;
} MCPState;
```

3.2.3.1.4.5 State Data Type

The state data type is used to construct flight trajectories and is pointed to by a flight plan after the flight plan has been passed through a flight trajectory calculation. The state data elements are defined below.

```
class StateCR
{
public:
        StateCR();      // Constructor
        ~StateCR();     // Destructor

  Waypoint* waypoint;
  double latitude;
  double longitude;
  double altitude;
  double time;
  Xyz    location;
  double vertical_speed;
  double cas;
  double ground_speed;
  double gross_weight;
  double mach;
  double ground_track;
  AtmosphericType  atmospheric;
  double x;
  StateCR* next;
  StateCR* prev;
};
```

Data Element	Description	Type / Units	Range
waypoint	pointer to the prior waypoint in the flight plan	Waypoint*	valid pointer into flight plan
latitude	Latitude (N is positive)	double / radians	(-π,π)
longitude	Longitude (E is positive)	double / radians	(-π,π)
altitude	Altitude	double / feet	(0,99999)
time	Time of aircraft presence at this location	double / minutes	(>0)
location	Unit vector of latitude and longitude described above.	Xyz / unit vector	
vertical_speed	Vertical speed at location	double / fps	valid double
cas	CAS at location	double / knots	≥ 0
ground_speed	Ground speed of aircraft at location	double / knots	≥ 0
gross_weight	Aircraft gross weight at location	double / pounds	≥ 0
mach	Mach number at location	double	≥ 0
ground_track	Ground Track angle relative to true North at current point	double / radians	(0,2π)
atmospheric	Atmospheric data at current location	AtmosphericType	
x	Along-track distance from PPOS at the current point.	double / nmi	≥ 0
prev	Pointer to prior point in trajectory. PPOS points to NULL.	stateCR *	NULL or pointer to valid data
next	Pointer to next point in trajectory. Final point in trajectory points to NULL.	stateCR *	NULL or pointer to valid data

3.2.3.2 Own-ship Trajectory Description

The CD&R function will receive an own-ship trajectory description associated with the own-ship flight plan. This trajectory description will use a list of states to define the flight trajectory. It is assumed that the trajectory can be linearly interpolated between points. The data structure defining the trajectory is defined below.

```
struct Ownship
{
  char AC_ID[10];
  struct Ownship* parent;
  bool vertical_manuever;
  struct State traj_sync[1000];
  struct State traj_Async[1000];
  double min_H;
  double max_H;
  struct IntruderNonPruneList *non_prune_intruder_list;
  struct AreaNonPruneList *non_prune_area_list;
  struct Conflict *conflict_list;
  struct AreaConflict *area_conflict_list;
  StateCR *state;
};
```

Data Element	Description	Type / Units	Range
AC_ID	String containing unique aircraft identifier	char *	ASCII string
parent	Pointer to "parent" trajectory. The original trajectory from which this one is derived	struct Ownship	NULL or valid pointer
vertical_maneuver	Boolean indicating if the trajectory contains a vertical maneuver away from the parent	bool	true/false
traj_sync	Array of states indicating the synchronized trajectory of the flight	State []	valid structures
traj_Async	Array of states indicating the non-synchronized trajectory of the flight	State []	valid structures
min_H	Minimum altitude of this flight trajectory	double/ feet	(0,99999)
max_H	Maximum altitude of this flight trajectory	double/feet	(0,99999)
non_prune_intruder _list	Pointer into list of intruders that have to be compared	struct IntruderNonPruneList*	valid structure pointer or NULL
non_prune_area_list	Pointer into list of area hazards that have to be compared	struct AreaNonPrune List*	valid structure pointer or NULL
conflict_list	Pointer to list of conflicts applicable to this trajectory	struct Conflict *	NULL or valid pointer
area_conflict_list	Pointer to list of area conflicts applicable to this trajectory	struct AreaConflict *	NULL or valid pointer

state	Pointer to a list of states defining the trajectory	StateCR *	NULL or valid pointer

The above makes use of the following data structures:

```
struct State
{
        double t;
        double x;
        double y;
        double z;
        double lat;
        double lon;
        double h;
};
```

Data Element	Description	Type / Units	Range
t	Time	double/seconds	>0
x	x-position Cartesian (NOT unit)	double / nmi	
y	y-position Cartesian	double / nmi	
z	z-position Cartesian	double / nmi	
lat	latitude	double / degrees	(-90,90)
lon	longitude	double / degrees	(-180.,180)
h	Altitude	double / feet	(0, 99999)

```
struct IntruderNonPruneList {
        struct IntruderNonPruneList *next;
        struct IntruderNonPruneList *prev;
        struct Intruder *intruder;//intruder in the NonPrune List
};
```

Data Element	Description	Type / Units	Range
next	Next intruder element in list	struct IntruderNonPruneList*	NULL or valid pointer
prev	Prior intruder element in list	struct IntruderNonPruneList*	NULL or valid pointer
intruder	Intruder trajectory to be compared against	struct Intruder*	Valid pointer

```
struct AreaNonPruneList {
        int place_in_list;
        struct AreaNonPruneList *next;
        struct AreaNonPruneList *prev;
        struct AreaHazard *area_hazard;
};
```

Data Element	Description	Type / Units	Range
place_in_list	Location in list of hazards	int	>0
next	Next intruder element in list	struct AreaNonPruneList*	NULL or valid pointer
prev	Prior intruder element in list	struct AreaNonPruneList*	NULL or valid pointer
area_hazard	Intruder trajectory to be compared against	struct AreaHazard*	Valid pointer

3.2.3.3 Own-ship State Description

The own-ship state description is defined by the first STATE element in the aircraft trajectory list. The first element is that whose prev element points to NULL.

3.2.3.4 Area Hazard Description

Area hazards are expected by the CD&R function to be received as one data structure containing all area hazards. These are described according to the following data structure.

```
struct AreaHazard
{
        char area_hazard_ID[10];
        struct AreaHazard* previous;
        struct AreaHazard* next;
        struct Node polygon[10];
        struct BoundaryPlane lat_haz_box_plane[4];
        double min_H;
        double max_H;
        double North_max;
        double South_max;
        double East_max;
        double West_max;
        SIDE_STRUCT side[10];
        int num_sides;
};
```

Data Element	Description	Type / Units	Range
area_hazard_ID	Unique identifier for hazard	char []	ASCII
previous	Prior area hazard in area hazard list	struct AreaHazard*	NULL or valid pointer
next	Next area hazard in area hazard list	struct AreaHazard*	NULL or valid pointer
polygon	Array of nodes defining area hazard	struct Node [10]	valid struct

	geometry		
lat_haz_box_plane	Array of calculated data to simplify area hazard calculations	struct BoundaryPlane[4]	valid struct
min_H	Minimum altitude of area hazard	double / feet	0,99999
max_H	Maximum altitude of area hazard	double / feet	0, 99999
North_max	Northern-most limit of area hazard	double / degrees	-90,90
South_max	Southern-most limit of area hazard	double / degrees	-90, 90
East_max	Eastern-most limit of area hazard	double / degrees	-180, 180
West_max	Western-most limit of area hazard	double / degrees	-180, 180
side	Array of data defining each area hazard side.	SIDE_STRUCT[10]	valid struct
num_sides	Number of sides defining area hazard	int	>0

The above uses the following data structures.

```
struct Node {
        double lat;
        double lon;
};
```

Data Element	Description	Type / Units	Range
lat	Latitude	double / degrees	(-90,90)
lon	Longitude	double / degrees	(-180,180)

```
struct BoundaryPlane {
        struct Point ref_point;
        struct Point normal_vector;
        struct Point position_vector;
};
```

Data Element	Description	Type / Units	Range
ref_point	reference point on boundary plane	struct Point	valid struct
normal_vector	Normal to the boundary plane of hazard box	struct Point	valid struct
position_vector	vector from boundary plane to aircraft current position	struct Point	valid struct

```
typedef struct Side {
        struct Point point1;
        struct Point point2;
        //normal to the plane that the boundary lies on
        struct Point norm;
} SIDE_STRUCT ;
```

Data Element	Description	Type / Units	Range

point1	Start point of side	struct Point	valid struct
point2	End point of side	struct Point	valid struct
norm	Normal to the plane that boundary lies on	struct Point	valid struct

```
struct Point {
        double x;
        double y;
        double z;
};
```

Data Element	Description	Type / Units	Range
x	x-position Cartesian (NOT unit)	double / nmi	
y	y-position Cartesian	double / nmi	
z	z-position Cartesian	double / nmi	

3.2.3.5 Intruder Aircraft Trajectory Description

Intruder trajectories are expected to follow a format defined below. Intruder trajectories are simpler than own-ship trajectories since certain own-ship data (e.g. flight plan information) is not available for intruders. Intruder trajectories are uniquely identified through their flight identifiers.

```
struct Intruder
{
        char AC_ID[10];
        struct Intruder *previous;
        struct Intruder *next;
        struct State traj_sync[MAX_SYNC_SIZE];
        struct State traj_Async[MAX_ASYNC_SIZE];
        double min_H;
        double max_H;
};
```

Data Element	Description	Type / Units	Range
AC_ID	Unique identifier for intruder	char []	ASCII
previous	prior intruder in list of intruders	struct Intruder *	valid pointer or NULL
next	next intruder in list of intruders	struct Intruder *	valid pointer or NULL
traj_sync	Array of synchronized trajectory points	struct State[]	valid structure
traj_Async	Array of trajectory points (not synchronized)	struct State[]	valid structure
min_H	Minimum trajectory altitude	double / feet	0,99999
max_H	Maximum trajectory altitude	double / feet	0,99999

3.2.3.6 Intruder Aircraft State Information

Intruder state information is obtained through the intruder aircraft trajectory. The first data element in the list of states is the most recent intruder aircraft state. The first element in the list of states is identified as the element whose previous element points to NULL.

3.2.3.7 (Deleted)

Section deleted

3.2.3.8 Boundary Constraints

It may be necessary for the CD&R function to accept boundary constraints. These represent constraints that occur at the point on the flight plan intercepting a specified boundary. Classes representing these constraints are defined below.

```
class BoundaryConstraint
{
public:
        BoundaryConstraint();    // Constructor
        ~BoundaryConstraint();   // Destructor

  BoundaryList* boundary;
  Restriction time_restriction;
  Restriction speed_restriction;
  Restriction mach_restriction;
  Restriction altitude_restriction;
  double max_latitude;
  double min_latitude;
  double max_longitude;
  double min_longitude;
};

class BoundaryList
{
public:
        BoundaryList();    // Constructor
        ~BoundaryList();   // Destructor

  double latitude;
  double longitude;
  Xyz xyz_location;
  BoundaryList* next;
  BoundaryList* prev;
};
```

Data Element	Description	Type / Units	Range
Boundary	List of boundary points on which constraint is to be applied	BoundaryList*	Pointer to valid list
time_restriction	Time restriction to be applied when the flight plan crosses this boundary	Restriction	Valid restriction
speed_restriction	CAS restriction to be applied when the flight plan crosses this boundary	Restriction	Valid restriction
mach_restriction	Mach restriction to be applied when the flight plan crosses this boundary	Restriction	Valid restriction
altitude_restriction	Altitude restriction to be applied when the flight plan crosses this boundary	Restriction	Valid restriction
max_latitude	Maximum latitude for this boundary	double / radians	$(-\pi/2,\pi/2)$
min_latitude	Minimum latitude for this boundary	double / radians	$(-\pi/2,\pi/2)$
max_longitude	Maximum longitude for this boundary	double / radians	$(-\pi,\pi)$
min_longitude	Minimum longitude for this boundary	double / radians	$(-\pi,\pi)$

Data Element	Description	Type / Units	Range
latitude	Latitude (N is positive)	double / rad	$(-\pi/2,\pi/2)$
longitude	Longitude (E is positive)	double / rad	$(-\pi,\pi)$
xyz_location	Unit vector from Earth's center representing location of corner point	Xyz	unit vector
next	Pointer to the next point describing the boundary Null is last point in list	BoundaryList*	NULL or valid pointer
prev	Pointer to the previous point describing the boundary. NULL is the first point in the list.	BoundaryList*	NULL or valid pointer

3.3 CD&R and Flight Rules

The CD&R function and the flight rules function exchange data as specified in this section. In order to preserve maximum flexibility in the flight rules, a large amount of information will be submitted to the flight rules. This provides access to the information required by the flight rules to determine useful resolutions.

3.3.1 Protocol

The CD&R and the flight rules function will operate serially. The flight rules function will be initiated by the CD&R function. The CD&R function will pass data to the flight rules function and subsequently wait for a response from the rules function. The flight rules function will pass data back to the CD&R function as a response. The flight rules are currently called through the following function call.

```
int ManeuverList * mainRules(struct Conflict*          conflict_list,
                             struct AreaConflict*      area_conflict_list,
                             const FlightPlan*         flight_in,
                             const struct Ownship*     traj_in)
```

3.3.2 Priority

All data received by the flight rules function is of equal priority.

3.3.3 Data Elements

The flight rules function will receive the following data from the CD&R function:

- Conflict information
- Own-ship flight plan
- Own-ship flight trajectory
- Own-ship aircraft state (included in Ownship data structure)

The flight rules will return to the CD&R function maneuver constraints to be followed by the own-ship. The return of maneuver constraints indicates that the own-ship must respond by displacing itself. The return of a NULL pointer in the place of maneuver constraints indicates that the own-ship is not expected to displace itself in response to the conflict.

3.3.3.1 Conflict Information

Traffic conflict information provided to the rules function is described through the following data structure.

```
struct Conflict
{
   struct Intruder *intruder;
```

```
    struct Conflict *next;
    struct Conflict *prev;
    struct State first_loss;
    struct State last_loss;
};
```

Data Element	Description	Type / Units	Range
intruder	The intruder that own-ship is in conflict with	struct Intruder*	Valid pointer
next	Next conflict in the conflict list	struct Conflict*	NULL or valid pointer
prev	Prior conflict in conflict list	struct Conflict*	NULL or valid pointer
first_loss	Copy of the first point in the own-ship flight trajectory point at which separation is lost.	struct State	Valid own-ship data
last_loss	The last point in the own-ship flight trajectory point at which separation is lost.	struct State	Valid own-ship data

Area hazard conflict information is provided to the rules function through the following data structure.

```
struct AreaConflict
{
        struct AreaHazard* area_hazard;
        struct AreaConflict* next;
        struct AreaConflict* prev;
        struct State first_intrusion_point;
        struct State last_intrusion_point;
};
```

Data Element	Description	Type / Units	Range
area_hazard	The area hazard that own-ship is in conflict with	struct AreaHazard*	Valid pointer
next	Next conflict in the conflict list	struct AreaConflict *	NULL or valid pointer
prev	Prior conflict in conflict list	struct AreaConflict *	NULL or valid pointer
first_intrusion_point	Copy of the first point in the own-ship flight trajectory point at which the trajectory enters the area hazard.	struct State	Valid own-ship data
last_intrusion_p	Copy of the point in the own-	struct State	Valid own-ship data

oint	ship flight trajectory point at which the trajectory exits the area hazard		

3.3.3.2 Own-ship Flight Trajectory

The flight trajectory is passed as described in the Section "Own-ship Trajectory Description."

3.3.3.3 Own-ship Flight Plan

The flight plan is passed as described in the Section, "Own-ship Flight Plan Data Description."

3.3.3.4 Own-ship Aircraft State

The aircraft state is passed as described in the Section, "State Data Type."

3.3.3.5 Description of Conflicting Hazards

The rules function will have access to the conflicting hazards through the hazard identifier provided with the conflict identifier. The data elements in area hazards are defined in the Section, "Area Hazard Description." The data elements in the traffic hazards are defined in the Section, "Intruder Aircraft Trajectory Description."

```
struct HazardDatabase {
        struct Intruder *intruder_list_head;
        struct AreaHazard *area_list_head;
        ACE_Thread_Mutex mutex_;
};
```

Data Element	Description	Type / Units	Range
intruder_list_head	List of traffic hazards as processed by CD	struct Intruder*	Valid pointer
area_list_head	List of area hazards as processed by CD	struct AreaHazard*	Valid pointer
mutex_	Mutex class used to lock access to the hazards database under multiple threads.	ACE_Thread_ Mutex	valid class

3.3.3.6 Maneuver Constraints

The Flight Rules function **may** return maneuver constraints to the CD&R function. When present, the maneuver constraints will be described as a list of maneuver arrays. Each maneuver array in the list will define a combination of maneuvers to be attempted by the resolution function (e.g., vector and speed). If the rules function does not return maneuver constraints, the returned maneuver list will point to NULL. Maneuver constraint data is defined below through two classes. Note that public data types were provided to provide a consistent transition from structures to classes with minimal re-coding.

```
class ManeuverConstraint
{
public:

  // Constructor
  ManeuverConstraint();
  ManeuverConstraint(char *, int , double , double , Waypoint *, Waypoint *);
  ManeuverConstraint(ManeuverConstraint*);

  //Destructor
  ~ManeuverConstraint();

  // Methods
  void setManeuver(char *, int , double , double , Waypoint * , Waypoint * );
  void setManeuverType(char *);
  void setDirection(int);
  void setStart(Waypoint *);
  void setEnd(Waypoint *);
  void setMaximum(double);
  void setMinimum(double);
  void setStartDesired(double);
  void setEndDesired(double);

  // Data
  char maneuver[15];
  char type[15];
  int direction;
  Waypoint* start_point;
  Waypoint* end_point;
  double max;
  double min;
  double start_desired;
  double end_desired;
};

class ManeuverList
{
public:
  // Constructors
  ManeuverList();
  ManeuverList(int, ManeuverConstraint *);
  ManeuverList(ManeuverList *);

  // Destructor
  ~ManeuverList();

  void addManeuver(int,ManeuverConstraint *);   //Method to add to list
  ManeuverList *jumpToEnd();                    //Jump to end of list

  // Class Data
  int number_of_maneuvers;
  ManeuverConstraint *maneuver;
  ManeuverList *prev;
  ManeuverList *next;
};
```

Data Element	**Description**	**Type / Units**	**Range**
maneuver	Maneuver degree-of-freedom	char []	"lateral" "global" "speed" "altitude" "time"

type	Specific type of maneuver dependent on degree-of-freedom	char []	if lateral:
			  general
			  vector
			  offset
			if global:
			  speed
			  altitude
			  climb_mach
			  climb_cas
			  descent_mach
			  descent_cas
			if speed:
			  temporary
			  permanent
			if altitude:
			  permanent
			  temporary
			  level-off
			if time:
			  time

direction	Direction of maneuver.
	  Positive maneuver if >0.
	  Either direction if ==0
	  Negative maneuver if <0
		int	valid int

start_point	Pointer to the first point in the flight plan at which the maneuver is allowed to begin.	Waypoint *	valid pointer into flight plan
end_point	Pointer to the last point in the flight plan at which maneuvers is allowed to begin	Waypoint *	valid pointer into flight plan
max	Maximum allowed displacement from the nominal flight plan	double / function of maneuver	
min	Minimum allowed displacement from the nominal flight plan (also refers to displacement in negative direction)	double / function of maneuver	
start_desired	Earliest desired starting time of maneuver	double / minutes	> 0
end_desired	Latest desired ending time of maneuver	double / minutes	> 0

Data Element	Description	Type / Units	Range
number_of_maneuvers	Number of elements in the array of maneuvers	int	>0
maneuver	Array of maneuvers to create a combined maneuver	ManeuverConstraint*	Valid pointer
prev	Pointer to previous combined maneuver to be attempted. A null pointer indicates the first maneuver array in the list.	ManeuverList *	NULL or valid pointer
next	Pointer to next combined maneuver to be attempted. A null pointer indicates the last maneuver array in the list.	ManeuverList *	NULL or valid pointer

3.4 CD&R and FMS Pre-processor

The role of the FMS pre-processor is to act as a gateway for data between CD&R and the FMS. This pre-processor was necessary since the data formats were not firmly established at the time of the design of CD&R, and since testing of the CD&R function required use of an existing FMS function. The role of the FMS pre-processor is to obtain a trajectory from flight plan data.

3.4.1 Protocol

The CD&R and the FMS pre-processing function operate serially. The FMS pre-processing function is initiated by a function call from CD&R. The CD&R function will pass data to the FMS pre-processing function and wait for a response from the pre-processing function. The FMS pre-processing function will pass trajectory data back to the CD&R function in response. The FMS pre-processing function is accessed through a function call with the following function prototype.

```
struct Ownship getTrajectory(FlightPlan , StateCR *);
```

Since multiple threads may be calling the above function, a mutual exclusion mechanism was required. This is achieved through a global variable that is shared by all threads wishing to call the getTrajectory function as defined below.

```
ACE_Thread_Mutex trajectory_mutex;
```

The above mutex is acquired at the beginning of all getTrajectory calls and released at the end. Should multiple threads wish to access the getTrajectory function, the acquire method will block until all other threads have released the mutex. This approach prevents global variables used by Fastwin functions from being modified during multi-threaded calls to the getTrajectory function.

3.4.2 Priority

Data passed across the CD&R to FMS pre-processing interface will all be at the same priority level.

3.4.3 Data Elements

The FMS preprocessor will receive flight plan and state information from the CD&R function and will return flight trajectory information to the CD&R function.

3.4.3.1 Flight Plan Description

Flight plan data is described in the Section "Own-ship Flight Plan Data Description".

3.4.3.2 State Description

Aircraft State data is described in the Section "State Data Type".

3.4.3.3 Trajectory Data Description

Trajectory data is described in the Section "Own-ship Trajectory Description".

3.5 CD&R and Constraint Manager

3.5.1 Protocol

TBD – currently expect CD&R to call CM and submit all the conflict, trajectory and flight plan information, this may need to be redefined.

The CD&R function passes data to the Constraint Manager function. Upon passing data, CD&R returns to a state of awaiting input from AOP and does not continue with conflict resolution. The Constraint Manager will subsequently complete its processing and submit modified trajectories to the CD&R function through the AOP calling function. The CM function will be initiated as a separate process to the CD&R function (TBD how implemented in ACE, for instance, CM could be running all the time and data passed from CD&R, or could be initiated by CD&R. Requirements for how to handle receipt of updated trajectory information when CM is running need to be defined.)

3.5.2 Priority

Data passed from the CD&R function to the Constraint Manager function will be at the same priority level.

3.5.3 Data Elements

TBD – Assumption that CM will simply receive a "data dump" from CD&R reflects lack of a specific approach to manage constraints. Many possibilities and ad hoc cases have been articulated, but a specific methodology has not yet been defined.

The Constraint Manager function receives the following data from the CD&R function.

Own-ship flight trajectory (type Ownship)

Own-ship flight plan (type FlightPlan)

Own-ship flight state (type StateCR)

Conflict description (type Conflict)

Area hazard description (type AreaHazard)

Intruder trajectories (type Intruder)

The Constraint Manager submits modified input information to the CD&R function through the same interface as the AOP calling function. As far as CD&R is concerned, that data is treated as any other conflicting flight plan.

3.5.3.1 Own-ship Flight Trajectory

The own-ship flight trajectory data elements are described in the Section “Own-ship Trajectory Description”,

3.5.3.2 Own-ship Flight Plan

The own-ship flight plan data elements are described in the Section “ Own-ship Flight Plan Data Description”.

3.5.3.3 Own-ship Flight State

The own-ship flight state data elements are described in the Section “ Own-ship State Description”.

3.5.3.4 Conflict Description

The description of conflict data is defined in the Section “Conflict Information”.

3.5.3.5 Area Hazard Description

The description of Area hazard data is defined in the Section “Area Hazard Description”.

3.5.3.6 Intruder Trajectories

The description of intruder trajectories is defined in the Section “Intruder Aircraft Trajectory Description”.

3.6 CD&R and User Input Function

3.6.1 Protocol

The CD&R function and the User Input Function will operate concurrently. Data will be passed asynchronously from the User Input Function to the CD&R function through the AOP calling function via messages. The protocol for these messages is defined in the protocol section of the CD&R and AOP Calling Function section.

3.6.2 Priority

The current implementation of messages between the AOP calling function and the CD&R function does not implement priorities on messages.

For future builds, one may consider the following:

Data from the User Input Function to the CD&R function will have priority levels assigned to them in the event of buffering between the functions. More recent data of

the same type will have priority over older data of the same type. User-supplied flight plans for manual resolution will have priority over other forms of data. User-supplied maneuver constraints will be second in the priority list of user-supplied data. Flight plans for provisional planning will be lowest in the priority list of user-input data. All other data types will have equal priority.

In the event of data queued between the User-Input Function and the CD&R function, the higher priority data jumps to the head of the queue for processing by the CD&R function.

3.6.3 Data Elements

The user input function will provide the CD&R function with the following information:

- User-supplied flight plans for manual resolution
- User-supplied flight plans for provisional planning (same format as above.)
- Resolution mode selection
- Request of next Flight Plan
- Resolution Accept
- User-supplied maneuver constraints
- User-supplied maneuver preferences

Some of these data requirements have been defined in prior sections. New data requirements are defined below.

3.6.3.1 User-Supplied Flight Plans

The data elements describing flight plans for both manual resolution and provisional planning are defined in the Section "Own-ship Flight Plan Data Description". These flight plans are supplied to the CD&R function through a MANUAL_FP_RES message (defined in the section "Messages from AOP to CD&R").

3.6.3.2 Resolution Mode Selection

Subsequent to manual input of maneuver constraints the user may desire that automatic resolution be performed on a potential conflict. An indication to the CD&R function that the user wishes to return to automatic resolution mode is required. The user input function will supply the indication to return to automatic resolution mode through FORCE_AUTO message as defined in the Section "Messages from AOP to CD&R".

3.6.3.3 Request of Next Flight Plan

Subsequent to completion of conflict resolution, multiple valid flight plans may be available for resolution. Even during the resolution process, conflict-free flight plans may already be available. The user will have the option to request that the next available conflict-free flight plan be submitted to the Crew Notification Function. The user input function will supply an indication to the CD&R function to send the next flight plan (in the ranked list of resolved flight plans) to the CNF. This indication will be through a REQUEST_RESOLVED message as defined in the Section "Messages from AOP to CD&R".

3.6.3.4 Resolution Accept

Upon user selection of a flight plan for conflict resolution, an indication will be sent to the CD&R function by the CNF, that a flight plan has been selected for resolution. This indication will be through an ACCEPT_RESOL message as defined in the Section "Messages from AOP to CD&R".

3.6.3.5 User-Supplied Maneuver Constraints

The data elements describing user-supplied maneuver constraints are defined in the Section "Maneuver Constraints". These are supplied to the CD&R function through a MANEUVER_CONSTRAINT message as defined in the Section "Messages from AOP to CD&R". The effect of the message is to switch the resolution mode to semi-automatic.

3.6.3.6 User-Supplied Maneuver Preferences

User-supplied maneuver preferences are to be selected from a menu of choices as defined below.

- Minimum time – the conflict-free maneuver meeting specified constraints is to be selected based upon minimum total time

- Minimum cost – the conflict-free maneuver that minimized a total cost will be selected. The total cost requires a specified cost index for the maneuver.

- Minimum fuel – the conflict-free maneuver that consumes the least fuel will be selected.

- Minimum constraints – the conflict-free maneuver that requires the least number of additional constraints will be selected. Of those maneuvers with an equal number of constraints, the flight plan with the least restrictive constraints will be selected.

- Minimum time away from original flight plan – the conflict-free maneuver that requires the least amount of time away from the original flight path (4D) will be selected. This is not a meaningful choice with a final RTA.

- Minimum distance away from original flight plan – the conflict-free maneuver that requires the least amount of distance away from the original flight path will be selected. This can be meaningful in 4D by considering the distance that the 4D paths are not equivalent.

This data will be sent from the CNF to the CD&R function through a MANEUVER_PREFERENCE message as defined in the Section "Messages from AOP to CD&R". However, the data within the message will contain the following data element.

enum ManeuverPreference{MIN_TIME, MIN_COST, MIN_FUEL, MIN_CONSTRAINT, MIN_TIME_AWAY, MIN_DIST_AWAY};

Data Element	**Description**	**Type / Units**	**Range**
man_preference	Type of maneuver preference for resolution	enum	MIN_TIME MIN_COST MIN_FUEL MIN_CONSTRAINT MIN_TIME_AWAY MIN_DIST_AWAY

3.7 CD&R and Long-term Optimization Function

3.7.1 Protocol

TBD

The long-term optimization function and the CD&R function operate serially. *(Note that the optimization function being described here only refers to that portion of the optimization function providing CD&R, upon request, with an optimized flight plan.)* The CD&R will initiate the long-term optimization function with a function call, and the CD&R will receive data from the long-term optimization function.

3.7.2 Priority

No priority is assigned to messages between the long-term optimization function and the CD&R function.

3.7.3 Data Elements

The long-term optimization function will receive a request for a long-term optimized flight plan and trajectory from the CD&R function. The long-term optimization function will return a long-term optimized flight plan in response to this request. In addition, the corresponding flight trajectory will be passed.

3.7.3.1 Long-Term Optimized Flight Plan

The Long-term optimization function will return a long-term optimized flight plan to the CD&R function. See the Section "Own-ship Flight Plan Data Description" for data descriptions.

3.7.3.2 Long-Term Optimized Flight Trajectory

The Long-term optimization function will return to the CD&R function, a long-term optimized flight trajectory corresponding to the above flight plan. See the Section "Own-Ship Trajectory Description" for data descriptions.

3.8 CD&R and Crew Notification Function

3.8.1 Protocol

The CD&R function and the Crew Notification Function will operate concurrently. Data will be passed asynchronously from the CD&R function to the CNF via the AOP calling function. The AOP calling function will receive messages from CD&R through a ReturnQueue as defined in the Section "Messages from CD&R to AOP".

3.8.2 Priority

Messages from the CD&R function to the Crew Notification Function may have priorities assigned to them based upon the urgency of the data to be presented. Data of higher priority are processed first by the CNF in the event of buffering between the CD&R function and the CNF. No specific messaging priority scheme has been built into the message queues.

3.8.3 Data Elements

The Crew Notification Function receives a collection of messages from the CD&R function. As the display mechanism for the AOP, the CNF is expected to receive the following data from CD&R.

- Conflict information
- Result of flight rules (maneuver constraints and decision to move)
- Result of resolution process including "best" candidate flight plan
- Notification that a conflict has disappeared

Note that hazard information (area and traffic) will be submitted to the CNF through a separate function. Rather than send processed data to the CNF, the initial implementation will supply the CNF with the raw data, allowing the CNF to process, filter and format the information for display.

3.8.3.1 Conflict Information

The Section "Conflict Information" defines the data requirements for conflict information being passed to the CNF by the CD&R (via the AOP calling function). This data will be contained in CONFLICT or CONFLICTED ***FLID*** messages with formats defined in the Section "Messages from CD&R to AOP". Note that the CONFLICT message is a response to the own-ship trajectory whereas the CONFLICTED ***FLID*** message is a response to a trial plan or manual resolution.

3.8.3.2 Result of Flight Rules

The CD&R will submit the "who moves?" flight rules decision to the CNF. This latter output of the flight rules function (a function call) will be translated to a THEY_MOVE or WE_MOVE message as defined in Section "Messages from CD&R to AOP" for submission to the CNF.

3.8.3.3 Resolution Result Output

Upon resolution of a conflict, or upon request by the User Input function, the CNF will receive a flight plan from the CD&R function through the AOP calling function. The format of the flight plan is defined in the section "Own-Ship Flight Plan Data Description". Receipt of a NULL flight plan will indicate that no further data is available. The output of resolutions is contained in a RESOLVED_FP or a LAST_RESOLVED_FP message as described in the Section "Messages from CD&R to AOP".

3.8.3.4 Conflict Disappeared

If a trajectory update or hazard update indicates that a conflict has disappeared, the CNF will be notified through updated conflict information via the AOP calling function. The absence of a specific conflict in the conflict information indicates that the conflict has disappeared. A NO_CONFLICT message is sent from the CD&R function to the AOP when no conflict is found and can be used to indicate the disappearance of a prior conflict. This messages is described in the Section "Messages from CD&R to AOP".